

CRAFT: A library for easier application-level Checkpoint/Restart and Automatic Fault Tolerance

Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein

Abstract—In order to efficiently use the future generations of supercomputers, fault tolerance and power consumption are two of the prime challenges anticipated by the High Performance Computing (HPC) community. Checkpoint/Restart (CR) has been and still is the most widely used technique to deal with hard failures. Application-level CR is the most effective CR technique in terms of overhead efficiency but it takes a lot of implementation effort.

This work presents the implementation of our C++ based library CRAFT (Checkpoint-Restart and Automatic Fault Tolerance), which serves two purposes. First, it provides an extendable library that significantly eases the implementation of application-level checkpointing. The most basic and frequently used checkpoint data-types are already part of CRAFT and can be directly used out of the box. The library can be easily extended to add more data-types. As means of overhead reduction, the library offers a built-in asynchronous checkpointing mechanism and also supports the Scalable Checkpoint/Restart (SCR) library for node level checkpointing. Second, CRAFT provides an easier interface for User-Level Failure Mitigation (ULFM) based dynamic process recovery, which significantly reduces the complexity and effort of failure detection and communication recovery mechanism. By utilizing both functionalities together, applications can write application-level checkpoints and recover dynamically from process failures with very limited programming effort.

This work presents the challenges addressed by the library, its design, and its use. The associated overheads are analyzed using benchmarks.

Index Terms—Application-level checkpoint/restart, automatic fault tolerance, User-Level Failure Mitigation (ULFM)



1 INTRODUCTION

THE ever increasing demand of more computational power continuously leads to the deployment of larger systems with more efficiency. After the halt in the increase of the processor frequency, the consistent growth in larger clusters is the result of the growing level of hardware parallelism. This results in a decrease of the mean time to failure (MTTF) with every new generation of large clusters. For example, the BlueGene/P system ‘Intrepid’ (debuted at #4 in the June 2008 top500¹ list) had a reported mean time to (hardware) interrupt of 7.5 days [1], whereas the more recent BlueGene/Q system ‘Sequoia’ (debuted at #3 in the Nov. 2013 top500 list) has a reported node failure rate of 1.25 per day [2]. This trend raises the concerns in the HPC community about the effective usability of clusters at the exascale level.

A program running on HPC systems can fail for many reasons, e.g., hardware and software faults, silent errors, Byzantine failures, etc. A study by El-Sayed et. al. [3] has found that 60% of all failures are attributed to either memory or CPU failures. Such failures, in addition to many

others, lead to process failure and eventually to the failure of the MPI-job as a whole.

There are many fault tolerance techniques to reduce the damage of faults to a minimum. These can be classified as one or a combination of the following four categories [4]: algorithm-based fault tolerance (ABFT), checkpoint/restart(CR), message logging, and redundancy. These categories vary by the faults they can detect and/or recover, their coverage, etc. The coverage of a fault tolerance technique is defined as the measure of its effectiveness [5]. None of the fault tolerance techniques can guarantee a 100% coverage of all possible failures, e.g., a technique against silent errors can not protect the application against hardware failures. Due to its characteristic properties, CR is the most widely used fault tolerance technique. The others, such as message logging and ABFT, also often use CR as a supporting component. Apart from fault tolerance, many HPC applications use CR to cope with other issues such as maximum walltime limit, which is present on almost all large production systems.

In parallel applications, CR services are categorized on two different levels: the mode of saving the process state, and the communication handling method during checkpointing.

The communication is handled either in an uncoordinated or a coordinated approach. In an uncoordinated checkpointing approach, each process takes its checkpoint independently without any notification to other processes [4]. Thus the communication channels are not flushed and

- F. Shahzad, M. Kreutzer, T. Zeiser, G. Hager and G. Wellein are affiliated with Erlangen Regional Computing Center (RRZE), University of Erlangen-Nuremberg, Martensstrasse 1, 91058, Erlangen, Germany. E-mail: faisal.shahzad@fau.de
- J. Thies is with the German Aerospace Center (DLR), Simulation and Software Technology, Linder Höhe, 51147, Cologne, Germany.

1. Top500 Website: <http://top500.org>

thereby are not in a consistent state. Despite its simplicity, this is generally not a preferred method due to its drawbacks in the restart phase and a large requirement of storage capacity [6], [7]. In contrast, a coordinated approach creates a checkpoint of all processes at logically the same time. Depending on whether the communication channels are blocked or not during checkpointing, the coordinated checkpointing is further divided into blocking and non-blocking approaches [8].

The methods for saving the state of the processes are categorized as follows depending on the level of transparency and location of implementation in the software stack [4]. 1) System-level: As its name suggests, this checkpoint is implemented on kernel level. Thus, the whole memory footprint of the application is checkpointed. 2) User-level: Implemented in the user-space, such a CR service captures the process state by virtualizing corresponding system calls to the kernel without being tied to the kernel itself. 3) Application-level: The user manually determines the data that needs to be checkpointed. This offers the possibility to save the minimum amount of data needed for a checkpoint and thus incurs minimum checkpoint overhead, which makes it a choice of application in many science and engineering packages. We therefore focus on the Application-level CR (ALCR) in this paper.

Even in the presence of an ALCR functionality in an application, the fail-stop failures lead to process/node failures(s), which eventually cause job abortion. Larger jobs have a much higher probability to encounter such failures, which means spending additional time in the queuing process of a typical cluster and then restarting the job. Therefore, besides saving the state of the job, it is essential to deal and dynamically recover from process failures in a well-defined manner, which we define here as ‘Automatic Fault Tolerance’ (AFT), and is the second focus of this work.

1.1 Challenges and Contributions

This subsection presents current challenges in the field of ALCR and dynamic process recovery of MPI applications.

Challenge 1: Many production engineering applications contain some sort of CR mechanism. This feature is not only motivated by the need to secure the application against faults, but also by the maximum continuous walltime limit of clusters. For this purpose, most applications implement their custom CR mechanism from scratch, which may be tedious and time-consuming, depending on the intricacy of the data structures. Advanced features like asynchronous checkpointing and node-level CR are therefore rarely implemented. We feel that there is a need to have an ALCR interface that is widely applicable, yet flexible enough to accommodate the complexity and versatility of user-defined data-types. The interface must be integrable into an existing production code without significant effort.

Challenge 2: A side effect of checkpointing is that it can degrade the application performance. Though it is impossible to eliminate its overhead, the ALCR service must offer the possibilities to reduce it so that it can be kept within acceptable bounds.

Challenge 3: In order to enable the dynamic process recovery in an application, the key ingredient is a fault

tolerant communication layer. Although there have been many efforts in the past to include fault tolerance into MPI, none of them have found their way into the official standard yet. Currently, the User-Level Failure Mitigation (ULFM) [9], [10] prototype implementation is under evaluation for inclusion in the MPI-4.0 standard. In order to give users the maximum flexibility to implement communication recovery, ULFM provides the key tools for failure detection, acknowledgment, and shrinking etc., but leaves the high-level implementation to the user. This way, the user is flexible in implementing the communication recovery process, e.g., shrinking recovery, non-shrinking recovery etc. However, as much of the recovery process of communicators follows a specific program flow, we believe that many details of this procedure can be hidden from the user by a generic recovery API that can be used for a wide range of applications.

To summarize, apart from technical challenges the scientific challenge here is to identify an as large as possible abstract class of applications that can be supported by a single CR library.

Contributions: This paper addresses the challenges described above in the form of a C++ library, CRAFT.

- 1) The first part of the paper deals with Challenge 1 and presents the design and implementation of CRAFT’s CR functionality, which provides an easy interface for making ALCR functions for a variety of data-types (POD, 1D- and 2D- POD arrays, MPI data-types, etc). In addition, the user can extend the library with arbitrary data-types. The extensibility of CRAFT is demonstrated by creating a simple class and showing the steps to make it a CRAFT checkpointable data-type.
- 2) For reducing the checkpointing overhead (Challenge 2), CRAFT provides a built-in asynchronous checkpointing mechanism. In addition, CRAFT also supports the Scalable Checkpoint Restart (SCR) library, which enables checkpoint storage and recovery at the node-level.
- 3) To address Challenge 3, we have fixed the fault detection method to exception handling, and hidden away many ULFM fault detection and communication recovery details behind an easy-to-use API (built on top of ULFM). The user can choose between shrinking or non-shrinking recovery. In addition, various functions are provided that are helpful for determining the data recovery strategy.
- 4) Two linear algebra applications namely Lanczos and Jacobi Davidson (JD) are used to showcase and analyze the overheads of CRAFT features. A weak scaling analysis of CRAFT’s CR module show that asynchronous threads can substantially reduce the overhead for PFS-level checkpoints, whereas the node-level checkpointing show a perfect scaling with minimal overhead. In addition, a breakdown of scaling analysis (up to 2560 cores, 128 nodes) of the AFT module is performed to highlight the better and poor scaling parts in the communication recovery process.

The frequency of the checkpoints is a critical parameter in this context, and is well-studied in the literature. A model for the optimal checkpoint interval can be found in [11].

1.2 Applicability

At this stage, it is important to explicitly identify the class of applications that can benefit from the CR solution presented

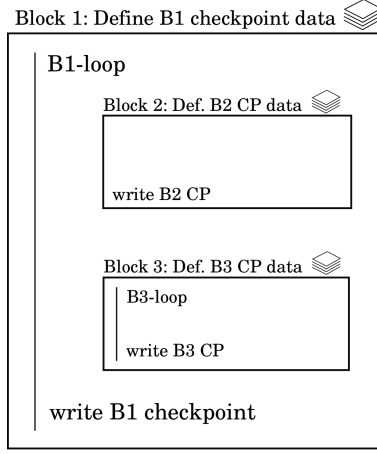


Figure 1: A sketch of a potential CRAFT application. The program consists of blocks where the checkpoint data of each block is defined at the start of the block. Each block may subsequently contain more sub-blocks and/or loop bodies. The checkpoints are written at the end of a block-/loop body.

in this paper. Our CR approach is applicable as long as the application flow does not contain unpredicted branches with sudden jump/break points. A sample control flow of a program that can use CRAFT’s CR solution is shown in Fig. 1. The program can be seen as combination of execution blocks, where a block may contain further blocks or a loop body. In order to create a checkpoint inside a loop, the data items which it entails, must be completely defined before entering the loop body. The checkpoints are written at the end of the block/loop body. This structure encompasses a large variety of applications, e.g., explicit time integration schemes for solving Partial Differential Equations (PDE) (1 loop), implicit time stepping with a Newton-Krylov non-linear solver (3 nested loops), continuation and linear stability analysis by solving a nonlinear PDE for a sequence of parameters and solving an eigenvalue problem at each steady state (a loop with two blocks inside, each containing a nested loop), etc.

On the contrary, such a CR solution is not feasible for applications that dynamically change the structure and size of the data-types to be checkpointed.

1.3 Outline

The rest of the paper is organized as follows. The design, implementation, and interface details of the CRAFT’s CR functionality are presented in Section 2. Section 3 presents the design and implementation of AFT in CRAFT. The details of benchmark applications, software environment and testbed systems are described in Section 4. The scalability as well as the overheads involved due to CR and AFT features of CRAFT are analyzed in detail in Section 5. In Section 6, a brief summary of the related work is presented. Section 7 presents a summary and concludes the paper.

2 CRAFT(I): CHECKPOINT/RESTART LIBRARY

This section presents the implementation details of the CR functionality of CRAFT.

```
#include <mpi.h>
#include <craft.h>
int main(int argc, char* argv[]){
    size_t n=5, iteration=1, cpFreq=10;
    double dbl = 0.0;
    int * dataArr = new int[n];
    // ===== DEFINE CHECKPOINT ===== //
    Checkpoint myCP ("myCP", MPI_COMM_WORLD);
    myCP.add ("dbl", &dbl);
    myCP.add ("iteration", &iteration);
    myCP.add ("dataArr", dataArr, &n);
    myCP.commit();
    myCP.restartIfNeeded (&iteration);
    for (; iteration <= 100 ; iteration++){
        // Computation-communication loop
        modifyData(&dbl, dataArr);
        myCP.updateAndWrite (iteration, cpFreq);
    }
    return EXIT_SUCCESS;
}
```

Listing 1: A simple application that shows the changes required (highlighted) to integrate CRAFT’s application-level CR functionality.

In this work, we define a ‘checkpoint’ as the collection of all data objects that are necessary to recover a particular stage of the program. A program can have multiple/nested checkpoints at various stages, each having a different checkpoint interval. The updated versions of a checkpoint are referred as ‘checkpoint-versions’ (CP-version). Each checkpoint can contain various data-types, such as plain-old-data (POD), POD-arrays, POD-multiarays, and even complex user-defined class objects. Moreover, any data object in a program can have parts which are not necessary for checkpointing. We say that a data-types is ‘CRAFT-checkpointable’, if its CR-related functions are linked to CRAFT.

The most common, standard and frequently used data-types (e.g. POD, 1D, and 2D POD-arrays, std::vector etc.) are contained in the default CRAFT-supported checkpointable data-types and can be used directly out of the box. In addition, the user can make any arbitrary data-type CRAFT-checkpointable by a simple extension mechanism, which essentially requires to implement its CR-related functions (discussed in detail in Section 2.2). The CRAFT library does not contain any communication synchronization protocol, as it relies on all processes making checkpoints at the same iteration, which enforces an implicitly consistent state of the application.

We start by looking at an example in which checkpoints are created using CRAFT. Listing 1 shows a simple iterative toy application in which ALCR functionality is added using CRAFT. The required CRAFT modifications have been highlighted. A Checkpoint object is created and all relevant data is added to it using the add() member function. In this example, a double element dbl, the iteration counter iteration and an integer array dataArr of length n are added. A pointer to each checkpointable object is saved. If the user has enabled the asynchronous checkpointing option, a copy of the added checkpointable object is created at this stage. Once all relevant data was

added, the `Checkpoint` object is committed, which means that no further data can be added. The application uses the original data for computation/communication in the usual manner. The `updateAndWrite()` member function updates and writes all checkpoint data at the iterations that match with the checkpoint frequency. The member function `restartIfNeeded()` determines if the program is restarted, in which case it reads the checkpoint.

2.1 Design

The design logic of the CR feature of the CRAFT library is explained in Fig. 2. The core part consists of a base class (`CpBase`) with three pure virtual functions called `read()`, `write()`, and `update()`. For each checkpointable data-type, a class is derived from `CpBase` which carries the implementation of its virtual functions. The user interacts with CRAFT mainly via a `Checkpoint` class. After creating a `Checkpoint` object, the user adds all necessary checkpointable objects into it by using `Checkpoint::add()` function. The `add()` function simply forwards the parameters to the appropriate specialized free function `addCpType()`, which instantiates the corresponding checkpointable data-type object (i.e, `CpPOD`, `CpPODArray` etc.) and adds it to a `std::map` container (`cpMap`) as shown in Fig. 2. This rather unique method of adding and bundling up the checkpoint data provides a neat API for the user to add ALCR with minimum code-modifications. The `Checkpoint::read()`, `-::write()`, and `-::update()` functions iterate through (`cpMap`) and call each checkpointable's corresponding functions. The `update()` function is used to update the copy of checkpointables and is required only in case of asynchronous checkpointing (further discussed in 2.3).

As the `add()` function is one of the most core functions of CRAFT, a brief description of its functionality and parameters is provided in the following.

`Checkpoint::add(string key, ...):` The `std::string key` argument is used to create the file name of each added object of a particular checkpoint, thus it has to be a valid file name. A brief description of few of the default supported checkpointable data-types is given below.

- **POD:** `add(string key, POD* dat)` can be used to add a POD element where `dat` could be a pointer to an `int`, `double`, `float`, `char`, `std::complex` etc. element.
- **POD 1D-Array:** `add(string key, POD* dat, int n)` can be used to add an array of POD elements, where `n` is the length of the array. It can also be used to add any contiguous data-chunk using a `char *`, in which case `n` specifies the chunk-size in number of bytes.
- **POD 2D-array:** `add(string key, POD** dat, int nDims, int* dims, string toCpData, int layout)` can be used to add a 2D POD array to a checkpoint. The dimensions of the array (number of rows and columns) are contained in the `dims` array. The `layout` describes the memory layout (row-major or column-major) of the 2D array. The user can furthermore specify, if only a given column or row is to be checkpointed using the `toCpData` string. For example, a value of "`x, 2`" indicates that all rows of the

2nd column should be checkpointed, whereas a value of "`CYCLIC, x`" indicates that rows must be checkpointed in a cyclic fashion.

- **MPI data-type:** `add(string key, void * buff, MPI_Datatype * data)` adds an object of any MPI derived data-type. For asynchronous writes, CRAFT uses `MPI_Pack()` to copy the data in a separate buffer, which is then saved asynchronously.
- **CpBase derived types:** `add(string key, CpBase * data)` can be used to add objects of data-types which were derived from `CpBase` by the user, and are therefore CRAFT-checkpointable.

In addition, the data-types of `std::vector`, `std::complex<POD>` and `std::complex<POD>*` array are also supported by default. CRAFT offers the following IO options for the default supported checkpointables. i) "`SERIALIO, BIN`" ii) "`SERIALIO, ASCII`" iii) "`MPIIO, BIN`". This option can be passed to `add()` functions via a `CpAbleProp` object.

The `Checkpoint::restartIfNeeded()` function checks for the existing checkpoints (from the previous application run) and, if found, reads the data. A detailed description of complete CRAFT interface, features, and options can be seen in the user-manual available at the CRAFT's repository [12].

2.2 CRAFT extension

In order to make an arbitrary new data-type CRAFT-checkpointable, the user may follow any of the following three methods. These methods differ based on the access rights to the corresponding data-type class.

- 1) If the user can modify the target data-type class, he can simply inherit CRAFT's `CpBase` class and implement its required CP-related virtual functions. In this way, the original class itself becomes CRAFT-checkpointable and the `Checkpoint::add()` member function can be directly used on its objects in the application.

We give an example using the simple class `rectDomain` shown in Listing 2. The class contains the three parameters `length`, `width`, and `val` that are of checkpoint interest. Listing 3 shows the extension methodology, where `CpBase` inheritance is added and its virtual functions are then implemented for the `rectDomain` class. The user can optionally create an extra copy of the data for asynchronous checkpointing and update the asynchronous copy in the `update()` function.

For implementing the `read()` and `write()` functions, the user is provided with the appropriate file name parameter from the back-end of the CRAFT library. This `filename` parameter is determined based on each checkpointable's path, name, version, and IO (MPI-IO or serial-IO) settings. It is therefore necessary that the user retains this critical information. The user can, however, derive more file names based on this in order to save the data in multiple files.

Instead of writing the POSIX or MPI-IO function explicitly, the user can employ the helper IO kit from CRAFT, which contains individual functions for writing/reading POD types, 1D-, and 2D-POD arrays with various IO

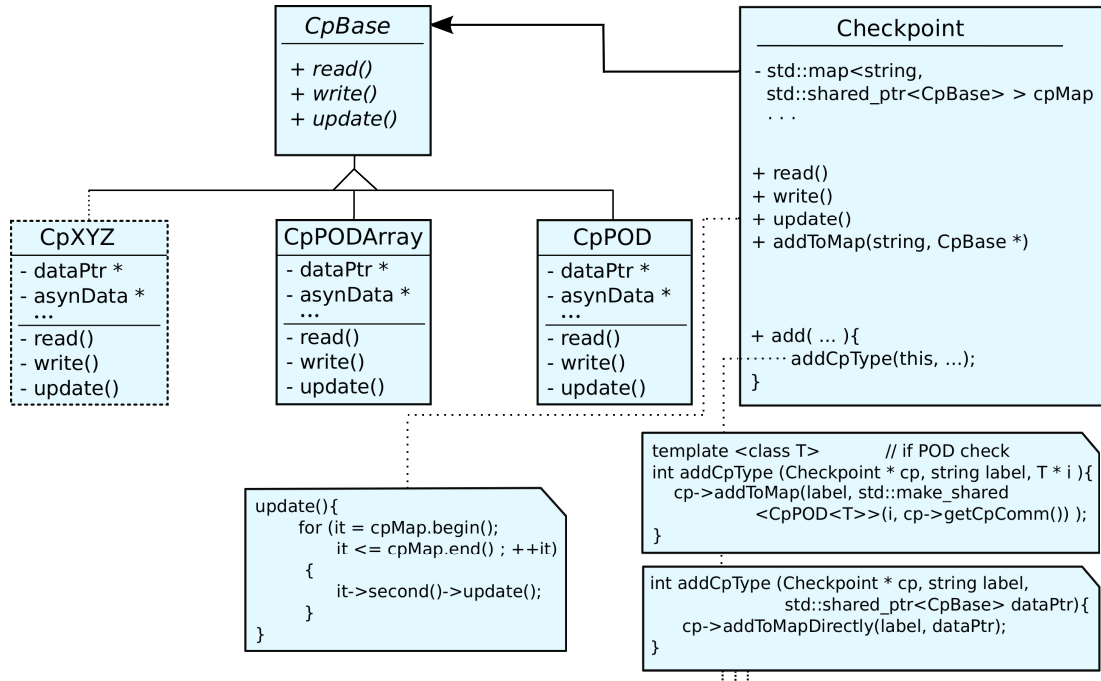


Figure 2: Design logic of the CRAFT library. The user creates a checkpoint by the adding checkpointable objects via the `Checkpoint::add()` member function, which eventually puts these objects in `cpMap`. The `Checkpoint::read()`, `write()` and `update()` methods iterate thought the `cpMap` and call each object's corresponding calls. Each checkpointable data-type class must be inherited from `CpBase` and must contain the implementation of its CP-related virtual functions (i.e., `read()`, `write()`, and `update()`).

```

class rectDomain{
public:
    rectDomain( const int length, const int width);
    rectDomain( const rectDomain &obj );
    ~rectDomain();
private:
    int length;
    int width;
    double * val;
};

```

Listing 2: A sample class `rectDomain`, whose objects a user may want to checkpoint in the application.

formats, e.g., MPI-IO, serial-binary, and serial-ASCII formats.

In the application, the user can now directly use the `add()` function on the `rectDomain` objects to include them in a checkpoint.

- 2) If the user for some reason cannot (or does not want to) modify the target class, a checkpointable wrapper class can be used which inherits CRAFT's `CpBase` class. The wrapper class holds a pointer to an object of the original class and contains the definition of its CP-related virtual functions. In the application, the user can now create this checkpointable wrapper class object (by using the original class object) and use the `add()` method to include it in the checkpoint.
- 3) In a similar approach as above, the user can also create the checkpointable wrapper class by inheriting CRAFT's `CpBase` as well as the original (e.g., `rectDomain` of Listing 2) class. This way the same objects can be used for normal computation/communication as well as for

```

class rectDomain: public CpBase
{
public:
    rectDomain( const int length, const int width);
    rectDomain( const rectDomain &obj );
    ~rectDomain();
private:
    int length;
    int width;
    double * val;
    double * asynVal;

    int update(){
        // copy *val array to *asynVal
    }
    int write(const std::string * filename){
        // write length, width, and *val in files
        static std::string fV = filename + "val";
        CRAFT_SERIAL_BIN_File_write_1D(fV, asynVal, length*width);
    }
    int read(const std::string * filename){
        // read length, width and *val from files
        static std::string fV = filename + "val";
        CRAFT_SERIAL_BIN_File_read_1D(fV, val, length*width);
    }
};

```

Listing 3: The CRAFT extension method 1: The `rectDomain` class of Listing 2 is changed to a CRAFT checkpointable data-type by inheriting from `CpBase` class and implementing the required virtual functions (`read()`, `write()`, and `update()`).

checkpointing purposes. A more refined solution would be to use different namespaces, each having the same class names but one being a CRAFT-checkpointable class in addition.

Helper IO Kit: As any arbitrary class eventually consists

of combinations of POD, 1D-, and/or multi-dimensional POD arrays, the CRAFT library offers a variety of useful POSIX- and MPI-IO functions for writing/reading POD, 1D- and 2D-POD arrays, etc. Separate IO functions for MPI-IO, serial-binary, and serial-ASCII are available for each variant. For example, the write functions for POD look as follows:

- `CRAFT_MPI_File_write_POD`
(string fn, POD* ptr, MPI_Comm comm);
- `CRAFT_SERIAL_BIN_File_write_POD`
(string fn, POD* ptr);
- `CRAFT_SERIAL_ASCII_File_write_POD`
(string fn, POD* ptr);

By providing a file name and appropriate arguments, the user can benefit from these functions while writing CRAFT extension functions for a new checkpointable class.

2.3 CR optimizations

The CRAFT library supports asynchronous checkpointing [13] and the Scalable Checkpoint Restart (SCR) library [14] as means to reduce the CR overhead.

The CRAFT library uses `std::future` and `std::async` to assign the asynchronous writing of checkpoints to a dedicated thread of each process. This method of checkpointing, in its naïve implementation, requires a separate copy of the data, so that writing and computation could take place simultaneously. For this purpose, all checkpointable default data-types in CRAFT create an additional copy if this option is enabled. The copy of the added data is updated from the original data in the `Checkpoint::update()` member function. In case of extending CRAFT for a new data-type, the user is responsible for creating a copy along with its `update()` method to benefit from this feature. Alternatively, the user can opt to perform the asynchronous writes without creating an additional data-copy. This can be enabled/disabled via the `CRAFT_WRITE_ASYNC_ZERO_COPY` environment variable. The user can then use `Checkpoint::wait()` to ensure the completion of the asynchronous write before modifying the data. As the IO routines can potentially use MPI routines, enabling this feature requires full threading support (i.e., `MPI_THREAD_MULTIPLE`) from the underlying MPI library. Using `CRAFT_ASYNC_THREAD_PIN_CPULIST` environment variable, the asynchronous thread locality (pinning) information can be provided, which is beneficial for gaining the maximum performance gain of asynchronous threads.

As an alternate measure to reduce the checkpointing overhead, the CRAFT library supports the SCR library [14]. The SCR library provides the user with the option for node level checkpointing. This can take one of three forms: local node level, partner level, or partner-XOR level. In case of a node failure, the partner level checkpointing allows to recover restart data from the failed node's neighbor. Apart from node level checkpoints, the less frequent checkpoints on PFS level can be made in order to enable recovery from multi-node failures. All necessary SCR-relevant changes, e.g., setup (`SCR_Init()`, `SCR_Finalize()`) and checkpoint calls (`SCR_Start_checkpoint()`, `SCR_Route_file()`,

and `SCR_Complete_checkpoint()`) are integrated into CRAFT. Once CRAFT is compiled with SCR, the node level checkpoints are automatically enabled for CRAFT-checkpointable data without any modification in the user application. The user can, however, disable SCR usage either partly (i.e., for specific checkpointable data using the `Checkpoint::disableSCR()` method) or completely (by using the `CRAFT_USE_SCR` environment variable). As our benchmark platform operates on a Torque resource manager, minor add-ons in SCR were required. Furthermore, the usage of the AFT feature (Sec. 3) in combination with SCR requires the SCR initialization with a fault tolerant communicator.

2.4 Multi-stage/Nested Checkpoints

As explained in Figure 1, the CRAFT library can be used to create multiple checkpoints in different code blocks of the program, where blocks may also be nested. However, careful attention must be paid while defining the checkpoints in a nested block. Firstly, all nested checkpoint definitions must be defined before entering the nested region. Secondly, depending on the application, it may be necessary to define a relationship between the inner- and outer level of checkpoints.

Listing 4 shows a pseudo-code, where the CL2 (inner-level checkpoint) is nested into the CL1 (outer-level checkpoint). The checkpoints are written at the end of each block. In most scientific applications, the results of the outer-level are updated using the results of the inner-level. However, depending on whether or not the results of an inner-level loop serve as the initial values for the next phase of inner-level loop calculations, the definition of nested checkpoints vary.

If the results of each CL2 cycle are to be carried as initial values of the next CL2 loop, there is no need to define a relationship between nested levels. Independent at which stage the failure occurs, the latest version of both checkpoint levels is read by CRAFT.

However, if the data and results of each round of CL2 iterations are independent from the previous CL2 iterations

```
Checkpoint CL1("CL1", FT_Comm);
Checkpoint CL2("CL2", FT_Comm);
CL2.subCP(&CP1); /* application dependent */
[...] // add L1data
[...] // add L2data
*****
CL1.restartIfNeeded();
for( L1iter —> nL1iter )
{
    /* L1 COMPUTATION COMMUNICATION */
    *****
    CL2.restartIfNeeded();
    for( L2iter —> nL2iter ) {
        /* L2 COMPUTATION COMMUNICATION */
        CL2.updateAndWrite(L2iter, L2cpFreq);
    }
    *****
    /* L1 COMPUTATION COMMUNICATION */
    CL1.updateAndWrite(L1iter, L1cpFreq);
}
*****
```

Listing 4: A pseudo-code example of nested checkpoints using CRAFT.

(e.g., each time initialed with zero before entering the CL2 loop), always reading the latest version of checkpoints will lead to inconsistency. Such a program requires that, in case of a restart when no CL2 checkpoints have been taken in the current CL1 iteration, the checkpoints from the previous round of CL2 iterations be left unread. However, if the failure happens after the first CL2 checkpoint is written, the latest versions of both (CL1 and CL2) checkpoint must be read. An example of such an application is a nonlinear solver (outer loop) with an iterative linear solver to compute corrections. The checkpoint of the linear solver (CL2) should only be read if it was written in the same outer iteration from which the application is restarted.

The CRAFT library solves this challenge by providing a special function, `Checkpoint::subCP()`, that defines a parent-child relationship between the nested checkpoints (as shown in Listing 4). By defining this relationship, all inner-level (CL2) checkpoints are invalidated as soon as an outer-level (CL1) checkpoint is written, thereby preventing the checkpoint data of the previous round of CL2 iterations from being read at the start of the next round of CL2 iterations.

Note that the `restartIfNeeded()` member function of the inner nested checkpoint (CL2 in the example) is called multiple times. Nonetheless the checkpoint is only read at the first instance of a restarted run. This method first checks the current CP-version of the corresponding checkpoint, which is 0 only at the first or restarted run of the program. A non-zero value of CP-version implies a successful successive nested loop run, thus it returns immediately without reading the checkpoint. The data consistency also requires the checkpoint frequency of each checkpoint level to be 1, except for the inner most checkpoint.

In case of asynchronous checkpointing with a combination of multi-stage or nested checkpoints, the user is responsible for the completion of previous/child checkpoints by calling `Checkpoint::wait()`. In the SCR library, each node level checkpoint must be self contained, i.e., the checkpoint files cannot contain data that spans multiple checkpoints. Due to this restriction, only one checkpoint in a nested checkpoint program can benefit from the SCR library, whereas the other checkpoints must be taken at the PFS level. Therefore, it is recommended to have high-frequency smaller checkpoints at the node level, and low-frequency larger checkpoints at the PFS level. The function `Checkpoint::disableSCR()` can be used to disable the use of SCR for a particular checkpoint object.

2.5 Signal-based checkpointing

The CRAFT library lets users take the checkpoint at any desired time during the program run, instead of at a fixed frequency. The `Checkpoint::updateAndWriteAtSignal(int iteration)` function can be used for this purpose. Whenever a checkpoint is desired, the user can trigger it by sending a signal via `kill -s SIGALRM <MPIEXEC_PID>` from the command line. Once this signal is caught by CRAFT's signal handler, it internally creates a synchronization barrier at the maximum iteration count of all processes. A checkpoint is updated and written only when all processes reach the synchronization iteration. This creates a checkpoint with all processes in a uniform state.

```
#include <mpi.h>
#include <craft.h>
int main(int argc, char* argv[]){
    ...
    int myrank, iteration = 0, cpFreq = 10;
    MPI_Comm FT_Comm;
    MPI_Comm_dup (MPI_COMM_WORLD, &FT_Comm);
    AFT_BEGIN (FT_Comm, &myrank, argv);
    *****
    double data = 0;
    Checkpoint myCP( "myCP", FT_Comm);
    myCP.add("data", &data);
    myCP.add("iteration", &iteration);
    myCP.commit();
    myCP.restartIfNeeded(&iteration);
    for (; iteration <= n; iteration++){
        /* Computation-communication */
        myCP.updateAndWrite(iteration, cpFreq);
    }
    ...
    AFT_END();
}
```

Listing 5: A simple application in which CRAFT's automatic fault tolerance feature has been added (highlighted) for communication recovery. The `AFT_BEGIN()` and `AFT_END()` functions form a fault tolerant communication zone.

This feature can be helpful if the status of a failing (unhealthy) node could be predetermined via monitoring different parameters, and thus saving processes states in a timely fashion. This implementation requires the underlying MPI runtime layer to recognize the signal received and pass it to all the running processes. As the `SIGALRM` signal is used to trigger the checkpoint, the user must make sure that it does not have a secondary use in the program. Its usage in CRAFT can be enabled/disabled at compilation time.

3 CRAFT(II): AUTOMATIC FAULT TOLERANCE

This section provides the implementation details of the AFT interface, which can be used for dynamic recovery of the application after process failure(s). Listings 5 shows an AFT enabled toy application. The AFT code modifications have been highlighted, whereas ALCR have been used for the data recovery. The AFT code first defines a communicator (duplicated from `MPI_COMM_WORLD`), which is used for all further communication. The most important additions in the code are the `AFT_BEGIN()` and `AFT_END()` macros that define an 'AFT zone'. The `AFT_BEGIN()` macro takes an MPI communicator and the input-argument string as inputs and returns the rank of the current process in the given MPI communicator. Within `AFT_BEGIN()`, an error handler is attached to the provided MPI communicator. The code inside the AFT zone is wrapped in a `try-catch` block surrounded by a `while()` loop that iterates until the `try` block's execution completes successfully (i.e., without throwing an exception). An exception is thrown at a process failure, in which case the communication repair mechanism is triggered in the `catch` block. Since the MPI Standard does not provide a fault tolerant communication interface, the core requirement for the AFT feature is ULFM-MPI.

3.1 ULFM-MPI

Though in its prototype stage, ULFM-MPI is the most powerful candidate for fault tolerant MPI functionality to be included in the MPI Standard. Its prototype implementation has attracted significant attention in the MPI applications development community [15], [16], [17], [18], [19].

Among other helpful functions the ULFM provides with `MPIX_Comm_revoke(MPI_Comm)`, and `MPIX_Comm_shrink(MPI_Comm, MPI_Comm)`, functions that can be used to detect process failures and recover the communicator. A detailed description of the complete ULFM-MPI API is available in [9]. The ULFM-MPI calls in combination with standard MPI functions such as `MPI_Comm_spawn()`, `MPI_Comm_get_parent()`, and `MPI_Intercomm_merge()` are showcased to implement a recovery strategy as examples by the ULFM-MPI working group at [20], which are used in the development of the CRAFT library's AFT module.

3.2 Communicator recovery

Using the basic ULFM interface, the developer is flexible to choose the failure detection and recovery methodology. In [21], Bland et. al. give an overview of application recovery strategies possible using ULFM. Once a particular recovery method is chosen, the order in which ULFM's failure detection and communication recovery functions should be called is quite unequivocal. We have leveraged this fact and came up with a very simple interface by abstracting away the details of all ULFM operations behind the `AFT_BEGIN()` and `AFT_END()` macro. An error handler is used for detection of failures, whereas for communication recovery, a shrinking or non-shrinking option can be chosen. Both of these recovery mechanisms have been adapted and further developed for CRAFT by the examples shown in [22]. Here a 'shrinking recovery' means that a new healthy communicator is created simply by removing the failed processes from the old communicator. On the other hand, a 'non-shrinking recovery' rebuilds the communicator by replacing failed process(es) by newly spawned ones. For the non-shrinking recovery, there can be two strategies of locality of the spawned processes. The environment variable `CRAFT_COMM_SPAWN_POLICY` can be either set to `REUSE`, which spawns the recovery processes on the same node as before, or to `NO-REUSE`, which spawns them on spare nodes, if available. The bookkeeping of reserve nodes (if available), failed nodes, as well as working nodes and processes is managed by CRAFT. The user can specify the recovery model and spawned processes locality information via environment variable parameters, (for detail, check [12]).

The following support functions are available to obtain useful information about the failed processes after the communicator has been recovered. They are particularly helpful to devise the data recovery strategy in case of a shrinking recovery.

- `bool AFT_isRecoveryRun()`: Returns `true` if the current run is a recovery run after failure recovery.
- `int AFT_getNumFailProc()`: Returns the number of failed processes.
- `int AFT_getFailedProcList(int *listFailed)`: This function takes an array (of

size equal to the number of failed processes) as an argument and updates it with the process numbers of the failed processes.

- `int AFT_getPastRank()`: Returns the rank of the current process in the communicator before failure recovery.
- `int AFT_getOrigNumProc()`: In case of successive shrinking recoveries, this function can be used to determine the number of processes in the communicator at the start of the application.

AFT's try-catch block and memory management:

Due to the presence of a try-catch block inside the AFT-zone, the proper memory management is a challenging task and requires careful attention. On the part of the user, the defined objects must follow the RAIL (Resource Acquisition Is Initialization) principle, which enforces the objects to clean up their memory usage as soon as they go out of scope. As of the memory allocated inside MPI, ULFM-MPI Standard [10] currently does not offer any automated mechanism to enforce cleanup of all internally allocated memory of a communicator in case it is suddenly revoked. However, the users are advised to call `MPI_Comm_free` on the communicators which they do not intend to use anymore, even if they contain failed MPI processes: "*..., this gives a high quality implementation an opportunity to release local resources and memory consumed by the object.*" [10]

In CRAFT, the old faulty communicator is freed at the end of the communication recovery process and a new communicator is returned to the user. As the old communicator is revoked as well as freed, all blocking/non-blocking communications of the new communicator cannot interfere with any outstanding blocking/non-blocking communications associated with the old communicator.

3.3 Data recovery

After rebuilding the communication structure, the next step is to recover the data of the lost processes. This step depends on the application as well as the communication recovery method and can take the form of CR, ABFT, etc.

CRAFT's CR functionality compliments its AFT feature. In the case of a non-shrinking recovery, the data recovery strategy is trivial. As the old surviving processes retain their rank numbers and newly spawned processes assume the ranks of the dead processes, each process can just read files based on their current rank. In a shrinking recovery, however, the ranks of processes may not be the same as before the failure. In this case, CRAFT offers both choices, either to read the checkpoint data based on the new ranks, or based on ranks before the failure (for detail, see [12]). In both of these cases, the user is responsible for explicitly managing and distributing the unread checkpointed data between the surviving processes.

4 EXPERIMENTAL FRAMEWORK

We have performed all benchmarks on the Emmy cluster² at RRZE. Equipped with 560 compute nodes, each having two Xeon 2660v2 "Ivy Bridge" chips (10 cores per chip + SMT) running at 2.2 GHz and 64 GB-RAM, the Emmy cluster has

2. Emmy cluster at the Erlangen Regional Computing Center (RRZE): <https://www.anleitungen.rrze.fau.de/hpc/emmy-cluster/>

the overall peak performance of 234 TFlop/s. The system has Infiniband interconnect with 40 Gbits/s bandwidth per link and direction. The parallel file-system (LXFS) has a capacity of 400 TB and an aggregated parallel IO bandwidth of more than 7000 MB/s.

4.1 Benchmark applications

We have used two sparse linear algebra applications to showcase CRAFT usage and evaluate the overheads involved.

Lanczos eigenvalue solver: The complete functionality of CRAFT is showcased using a Lanczos solver. The Lanczos algorithm is an iterative method for finding some eigenvalues of a sparse matrix. We use it to find the minimum eigenvalues of a test matrix. The pseudo-code of the Lanczos algorithm is shown in Algorithm 1. Each iteration calculates the new Lanczos vectors, α , and β . The approximated minimum eigenvalues are then calculated using the QL method and checked against the convergence criterion. In order to have a deterministic runtime, we fix the number of iterations in our benchmarks. However, in a practical code, a residual test with a desired tolerance of calculated eigenvalues is used as the convergence criterion to abort the iteration loop. The checkpoint data mainly consists of the Lanczos vectors, α , and β values etc. For this purpose, CRAFT is extended for the vector data-types of the utilized numerical linear algebra library GHOST [23].

Algorithm 1 The pseudo-code of the Lanczos algorithm for finding eigenvalues of a matrix A .

```

for j:=1,2, ..., numIter do
  function LANCZOS-STEP
     $\omega_j \leftarrow A\nu_j$ 
     $\alpha_j \leftarrow \omega_j \cdot \nu_j$ 
     $\omega_j \leftarrow \omega_j - \alpha_j \nu_j - \beta_j \nu_{j-1}$ 
     $\beta_{j+1} \leftarrow \|\omega_j\|$ 
     $\nu_{j+1} \leftarrow \omega_j / \beta_{j+1}$ 
  end function
   $CalcMinimumEigenVal()$ 
end for
```

Jacobi-Davidson eigenvalue solver: This method of eigenvalue solver is popular in e.g. quantum physics and chemistry to determine a set of eigenpairs (λ_i, ν_i) for a large sparse matrix $A \in \mathbb{R}^{n \times n}, \mathbb{C}^{n \times n}$. The basic JD method consists of two principles. The first principle (suggested by Davidson) is to solve the eigenvalue problem with respect to a given subspace V_m spanned by orthonormal basis vectors (ν_1, \dots, ν_m) , which leads to the condition: $V_m^* A V_m s - \theta s = 0$. The solved Ritz values and vectors $(\theta_j^{(m)}, u_j^{(m)})$ of this equation give the eigenpairs of the A with respect to the subspace spanned by V_m , i.e., the projected eigenvalue problem. To improve the subspace, a linear system (correction equation) is solved approximately to find the orthogonal correction of t for $u_j^{(m)}$ such that $A(u_j^{(m)} + t) \approx \lambda(u_j^{(m)} + t)$. This basic idea stems from the Jacobi method. Details and theory of JD algorithm can be found in [24]. State-of-the-art implementations use Krylov methods to solve the correction equation, and locking and restart for computing multiple eigenpairs [25], which leads to the JDQR method.

Depending on the size of the system, number of eigenvalues sought, and required accuracy, etc., the runtime of JD algorithm can be significant. Therefore we used CRAFT to introduce ALCR in PHIST implementation of JD algorithm. The PHIST (Pipelined Hybrid Parallel Iterative Solver Toolkit) library [26] provides implementations of and interfaces to block iterative solvers for sparse linear and eigenvalue problems and supports multiple backends, e.g., Trilinos, PETSc, GHOST, etc. PHIST implements a block version of the JDQR algorithm that improves the efficiency of the algorithm on modern hardware [27].

In the JDQR algorithm, the checkpoint data mainly consists of the subspace block-vector (V_m above) that is extended in each iteration by one or more vectors. Whenever m reaches a value m_{max} , the space is compressed to contain m_{min} vectors, including the already converged ('locked') eigenspace. This 'restart' stage of the algorithm is ideal for checkpointing as it produces the minimum checkpoint size. Other required quantities (e.g. $A \cdot V_m$ and $V_m^T A V_m$) are recomputed after a restart.

The CRAFT library is extended to make the PHIST vector data-types checkpointable (by method # 2 of Section 2.2). The implementation of the CRAFT-enabled JD algorithm is available at [28].

In our benchmarks applications, we use two matrices from the field of quantum-mechanics. The first matrix 'spinSZ<L>' represent the Hamiltonian of the Heisenberg spin chain model for L spins. Its dimensions grow exponentially with L , and the number of nonzeros per row varies significantly. The second matrix 'Graphene-<K>-<L>' represent a lattice of $K \times L$ graphene sheet. Graphene is the blueprint for quasi 2D materials with many distinctive characteristics and has prospective application areas in nanotechnology and nanoelectronics. The matrix dimensions are $K \times L$ and its sparsity pattern is very regular. These matrices can be generated from within the PHIST and GHOST libraries, which saves the expensive step of reading the matrix from the PFS.

4.2 Underlying MPI Implementations:

For all AFT-benchmarks, the ULFM-MPI release 1.1 version is used. The ULFM-MPI extensions features are built on top of Open MPI version 1.7.1. As described in 2.3, the asynchronous checkpointing optimizations require the MPI library to support `MPI_THREAD_MULTIPLE`. The Open MPI version 1.7.1 is recognized to have erroneous behavior under `MPI_THREAD_MULTIPLE` [29]. Therefore, in order to benchmark the CR optimizations (in Sec. 5.1), Intel MPI version 5.1 is used.

4.3 Benchmark categorization and fault model

The benchmarks are divided into three parts. Firstly, the benchmarks are presented that only evaluate the CR functionality of CRAFT. This constitutes of application runtime without CR, synchronous PFS-CP, asynchronous PFS-CP, and node level CP. Secondly, the ULFM-MPI is used to perform the micro-benchmarks to evaluate the process recovery overhead (AFT) and its scaling behavior. Thirdly, we make use of CR and AFT together to showcase their usage in real-world applications.

Fault and recovery model: In this paper, we focus on fail-stop failures, i.e., failures that cause a process to fail permanently. These processes become nonresponsive to any communication request, thus they can be detected during the following communication request involving the failed process by the ULFM failure detection mechanism. We usually simulate the failure of a complete node crash by killing all processes on a particular node via `pkill -9 <program>`. However, in a few benchmarks (in order to have a deterministic re-computation overhead), we have injected the “process failures” at a predetermined iteration from inside the program.

For the application benchmarks, we use non-shrinking communication recovery method in combination with CRAFT’s CR functionality as the data recovery approach. This combination eliminates the need to redistribute the domain (as in the case of shrinking recovery). Thus, the processes can make process-local checkpoints. This, in turn, enables the use of SCR for node-level checkpoints, which reduces the overhead.

5 RESULTS AND EVALUATION

This section shows performance results for simple benchmarks and the application scenario described above. In a fault tolerance library/tool, it is important to explicitly show the overhead faced by the application due to the presence of FT capabilities. Within the scope of this work, these overheads can be categorized into the following groups.

- Checkpoint and restart overhead (OH_{cp} , OH_{res}): An application faces checkpoint overhead even in the absence of failures. A failure recovery imposes additional restart overhead. The restart overhead includes the data re-initialization as well as reading the data from the checkpoint. Optimization methods such as asynchronous and node-level checkpointing aim at reducing this overhead.
- Communication recovery overhead (OH_{rec}): This overhead comprises only the time it takes to recover the broken communicator. The overhead varies depending on the communication recovery policy.
- Recomputation overhead (OH_{redo}): This overhead depends on the point in time between two successive checkpoints when the program faces a failure. It can be influenced by choosing an appropriate checkpointing interval. In addition, ABFT techniques can be used to accelerate the recovery process [30].

5.1 Checkpoint/restart benchmark

We employ Lanczos and JD applications to determine the impact of CR-overhead (OH_{cp}) using the optimization techniques described in Sec. 2.3. The Lanczos benchmark is run with the following parameters: num. of iterations=3000, checkpoint frequency=500, matrix: Graphene-30000-30000, matrix num. of rows & columns = $9.0 \cdot 10^8$, number of non-zeros = $11.7 \cdot 10^9$, global checkpoint size ≈ 14.4 GB; Whereas the JD benchmark has following parameters: matrix num. of rows & columns = $1.6 \cdot 10^8$, number of non-zeros = $2.6 \cdot 10^9$, num. of sought eigenvalues: 20, convergence tolerance: $1.0 \cdot 10^{-12}$, matrix: spinSZ30 min. subspace block-vector size (m_{min} , number of checkpointed vectors) = 28, max.

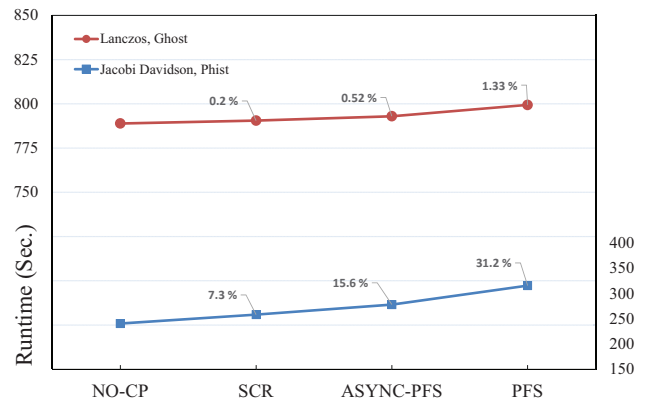


Figure 3: An overhead comparison for the Lanczos and JD benchmarks using three checkpointing methods of CRAFT, namely, node-level checkpointing with SCR, asynchronous PFS, and synchronous PFS checkpoints. The overhead for each checkpoint case is shown in percentage. (num. nodes=128, num Procs.=256, Intel MPI).

subspace block-vector size (m_{max}) = 48, global checkpoint size ≈ 32 GB, num. of checkpoints= 10.

Figure 3 shows this comparison of CRAFT supported checkpointing techniques on 128 Emmy nodes for the Lanczos and JD benchmarks. The baseline for the overhead is defined by the application runtime without any checkpoints (“No-CP” case). For both applications, the synchronous PFS checkpoints incur the highest overhead among the three techniques. The asynchronous PFS checkpointing technique significantly reduces the overhead, whereas the least overhead is caused by the node-level SCR checkpoints. It must be noted that the absolute overhead depends on multiple factors, such as checkpoint size and frequency, number of nodes, PFS and network bandwidth, etc. However this relative comparison indicates a qualitative difference in the efficiency of different checkpointing techniques. The cost of checkpointing for the Jacobi-Davidson method is significantly higher because of the larger checkpoint size and higher checkpoint frequency. Depending on the MTTF of the target machine one could write the checkpoints only at every k ’th restart to reduce this overhead in practice.

The weak scaling behavior of these techniques is analyzed by Lanczos application shown in Fig 4. In this weak scaling, the checkpoint size increases linearly with the problem/matrix size (indicated in upper x-axis). The benchmark reiterates the fact that synchronous PFS checkpointing technique scales very poorly. In comparison, an asynchronous PFS checkpointing techniques shows an improvement in overhead reduction. The neighbor-level checkpoints with SCR show minimum overhead to the application and scales perfectly.

5.2 Communication recovery (AFT) overhead

We analyze the scaling behavior of communication recovery (i.e., AFT) overhead (OH_{rec}), without any influence from the highly application-dependent checkpointing overhead. A simple benchmark is used for this purpose, in which

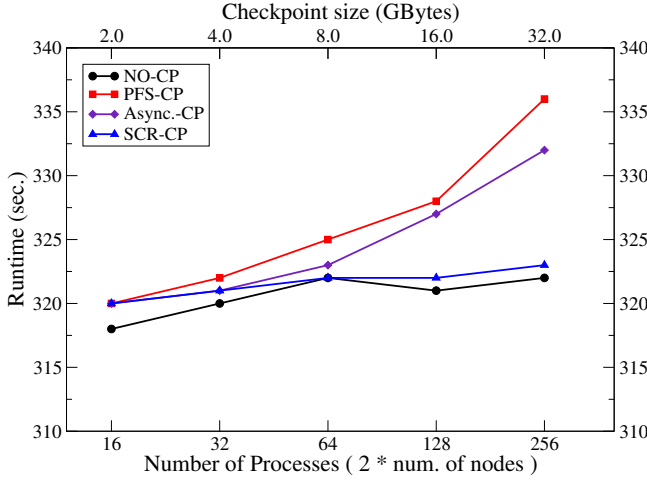


Figure 4: Weak scaling behavior of Lanczos application using different checkpointing techniques supported by CRAFT. (num. of iterations=200, num. checkpoints=4).

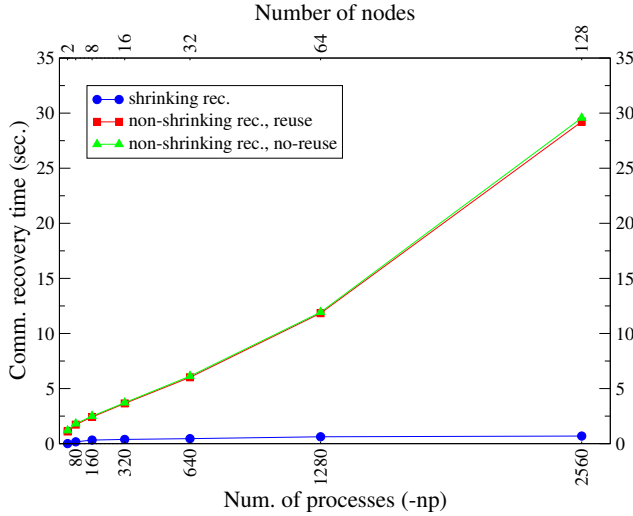


Figure 5: The scaling of MPI communication recovery time w.r.t. the number of processes (20 processes per node) for Shrinking, Non-Shrinking with reuse of failed nodes, and Non-Shrinking with no-reuse of failed nodes.

an `MPI_Barrier` is repeatedly called in a loop inside an AFT zone. Once process(es) fail, all other processes of the communicator are notified and follow recovery routines as explained in Sec. 3.2.

Figure 5 shows the scaling of the communication recovery overhead on up to 2560 processes (with 20 processes per

Description	Time(sec.)
Communicator revoke + shrink	0.34
Generate processes-spawn info.	0.23
Spawn + merge	26.10
Redistribute proc. ranks	1.39
Resource management	0.68

Table 1: Breakdown of the communication recovery phase with ‘non-shrinking, no-reuse’ recovery policy with 2560 processes on 128 nodes.

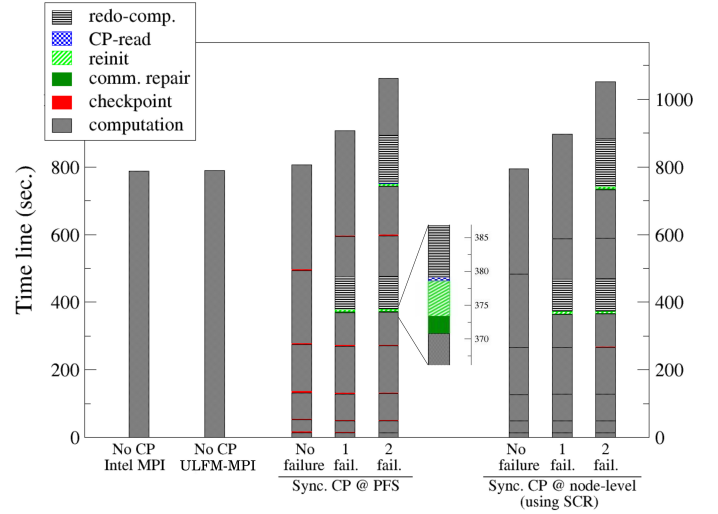


Figure 6: Various checkpoint and failure-recovery scenarios of the Lanczos benchmark using 128 Emmy nodes (256 processes). The average communication recovery time in process failure cases is 2.6 seconds.

node) using different recovery modes. The shrinking recovery mode shows the best scaling behaviour with very little influence from the number of processes involved. The recovery time is ≈ 0.51 seconds for a total of 2560 processes running on 128 nodes. The non-shrinking recovery overhead, however, increases linearly with respect to the number of processes. Table 1 shows the recovery overhead breakdown. The major part of it comes from the blocking operations of `MPI_Comm_spawn()` and `MPI_Intercomm_merge()`. The rest of the overhead comes from creating spawn information (via `MPI_Info_set()`), reassigning spawned processes the same identity as failed processes, and managing the available recovery nodes. The node ‘reuse’ policy for spawned processes is slightly cheaper than the ‘no-reuse’ policy. This is due to extra steps involved to manage the record of available recovery nodes.

Once a node has a hard failure, it is likely to encounter failures again. Thus, it is advisable not to use the same node again, whenever possible. In the following, we only use the ‘no-reuse’ policy of the non-shrinking recovery.

5.3 Automatic Fault Tolerant application

We now make use of both the CR and AFT parts of CRAFT to showcase their use with the Lanczos (Sec. 4) application. We use the same application parameters as in Sec. 5.1. The ULFM-MPI library is used here for evaluating the AFT feature. The failures are introduced from inside the application at the mid-point of two successive checkpoints.

Figure 6 shows the runtime of the Lanczos benchmark on 128 nodes (256 Processes, 10threads/process) in various cases. The first case, ‘No CP’, shows the runtime of 790 sec. without any checkpoints and failures. This constitutes the baseline for all other cases in this benchmark. This baseline can be compared with the baseline of the benchmark presented in Sec. 5.1 to evaluate the overhead introduced by ULFM-MPI, which is almost negligible (0.13%). Depending on the location of checkpoint storage (PFS, node level), the

benchmarks are grouped into two categories. The first group ‘Sync. CP @ PFS’ shows the program timeline with checkpoints at PFS. Each PFS checkpoint introduces an overhead of around 2 sec., compounding a total overhead (OH_{cp}) of $\approx 1.26\%$. The second group of bars ‘Sync. CP @ node-level’ shows the timeline with neighbor-level checkpoints using SCR. Here each checkpoint causes an overhead of ≈ 0.9 seconds. The main overhead appears in failure recovery cases (2nd and 3rd bar of each group), where the major part of the overhead is spent on the re-computation of lost work (OH_{redo}). Independent of the checkpoint location, the communication repair (OH_{rec}) takes ≈ 2.6 seconds.

Note that these benchmark results are presented to showcase the behavior of CRAFT in a particular (albeit typical) application setting on a particular machine. As such, they cannot be generalized to other scenarios without actually re-running the tests.

5.4 Additional CR extensions

In addition to the standard types, the CRAFT library is extended and used in the following sparse linear algebra solvers/libraries: GHOST [23], PHIST [26], BEAST [31].

6 RELATED WORK

The research and development of CR services can be categorized based on the level of transparency and software layer implementation.

A few libraries such as Open MPI [4], [32] and LAM/MPI [33] (using BLCR [34]) and MPICH-V [35] (using Condor [36]) provide a very transparent way to make system-level checkpoints. Libckpt [37] was a CR tool that offered a hybrid between system and user-level checkpointing. Its optimizations included incremental and forked checkpointing. The functionality to exclude certain parts of memory made it possible to minimize the overall checkpoint data volume.

With a strong focus on portability, tools like PC^3 [38] (based on C^3 [39]), CPPC [40] and Porch [41] support semi-automatic ALCR services via pre-compiler and/or code analysis approaches. The user must provide directives at the potential checkpoint locations, which gets instrumented by the pre-compiler during source-to-source compilation stage. Unlike the classic ALCR approaches, the compiler-assisted ALCR services are less transparent, and save the entire state of the program that creates checkpoints almost as large as system-level checkpoints. CPPC also offers the explicit definition of data via variable registration for the application-level checkpoints. These variables are registered as memory segments of basic data-types. The checkpoints are then written in a portable (platform-independent) format. The pre-compiler assists the user in registering the restart-relevant variables based on a liveness analysis and also modifies the flow-control of the application.

The Fenix library presented in [42] is similar to CRAFT and touches the same two aspects of fault tolerance, the data recovery (as an ALCR service) and process recovery (based on ULFM). Similar to CRAFT, the user must register the desired data to be checkpointed. Various forms of data can be combined to form a group. The storage of data is,

however, different in both libraries. Whereas Fenix focuses on checkpointing 1D-arrays whose copies are saved on neighboring nodes as checkpoints, CRAFT’s CR solution has a strong focus on file system checkpoints, and can leverage the SCR [14] library to create neighbor-level checkpoints. The file system based checkpoint solution adds flexibility to CRAFT as the user can write data using parallel I/O (MPI-I/O) or serial I/O (ASCII/binary) formats. This also enables users to use CRAFT as a generic I/O library as well. Apart from the basic out-of-the-box checkpointable data-types (e.g., POD arrays), the CRAFT library provides an extension mechanism for adding arbitrarily complex classes to its checkpointables.

The dynamic process recovery of CRAFT is largely similar to FENIX in its functionality, but carries operational variations. For example, the recovery behavior (shrinking or non-shrinking) can be controlled at run-time via environment variables. CRAFT also manages the hardware information of the job so the user can control whether to reuse the failed node again (for process spawning), or use a spare node instead. If the user decides to opt for a shrinking recovery, CRAFT offers a helpful API to extract the information about the failed processes, so that the user could make a strategy for redistributing the application domain.

The AFT [43] is a relatively new tool that semi-automatically inserts CR functionality into the code based on code analysis and user input. Another Open MPI [44] ALCR service is called SELF [32], which uses callback functions to activate checkpoints. The actual writing of the data is the responsibility of the user. At a desired stage, the user can initiate the checkpoint from outside the program, which starts Open MPI’s coordinated CR-service followed by the calls to the user’s implemented functions.

CRAFT’s ALCR service is also partly similar to the OpenMPI-SELF CR service in that it requires the user to provide I/O routines in case of an arbitrary data-type. However, CRAFT provides out-of-the-box I/O methods for most basic data-types. Once a new data-type is made CRAFT-checkpointable, it can avail the underlying optimization techniques like asynchronous and node-level checkpointing.

Before ULFM [9], several efforts revolved around fault tolerance on the MPI communication level. FT-MPI [45] (later known as HARNESS [46]) was the first widely known activity in this regard. FT-MPI offered three different failure recovery modes: SHRINK, BLANK, and REBUILD. The fault detection and communication recovery was part of FT-MPI, which minimized the code changes in the application. However, the project is no longer maintained. MPI/FT [47] was a similar effort which provided process failure recovery for two application models: master-worker and Single Program Multiple Data (SPMD). In case a worker process failed in the master-worker approach, the MPI library would notify the master, which would relaunch the worker. In the SPMD applications, it repaired the `MPI_COMM_WORLD` by replacing the failed processes and reading the checkpoint. However, it required the SPMD applications to have synchronous loops on all processes.

7 SUMMARY

CRAFT is a library that provides two important building blocks for creating a fault tolerant application. Its checkpoint/restart (CR) part is an extendable library base, using which the application-level checkpoint/restart functionality can be easily introduced in the programs with minimal modifications. The most frequently used data-types (POD, 1D-, and 2D-POD arrays, MPI derived data-types, etc.) are part of CRAFT by default and can be used out of the box. Furthermore, user defined data-types can easily be made CRAFT-checkpointable. To reduce the checkpointing overhead, CRAFT offers a built-in asynchronous checkpointing mechanism and also supports the Scalable Checkpoint/Restart (SCR) library.

Based on ULFM-MPI, the automatic fault tolerance (AFT) part of the library provides an easier interface for the dynamic process recovery in case of process failures. CRAFT hides many details of the process failure detection and communication recovery process, and enables the user to choose between a shrinking or a non-shrinking recovery of the failed communicator with marginal effort. The current AFT implementation includes support for Torque and SLURM job managers.

In this work, we have presented the design, implementation details and usage of CRAFT. We have also shown how its CR functionality can be extended for a user defined data-type. Two applications from sparse linear algebra have been taken (Lanczos and Jacobi-Davidson) to showcase its usage and analyze the related overheads. The weak-scaling benchmark shows clear improvement of CRAFT optimizations (node-level and asynchronous checkpointing) over usual synchronous PFS checkpointing.

The AFT benchmarks in real applications revealed that the communication recovery overhead itself remained in an acceptable range. The major part of the total overhead comes from the re-computation of the lost work, which depends on the point in time between two successive checkpoints when the failure occurs.

The scaling behavior of AFT was tested on up to 2560 processes on 128 nodes using a micro-benchmark. It showed that the newly added extended functions of ULFM-MPI behave well at scale. However, the majority of the recovery overhead was traced back to the poor scalability of `MPI_Comm_spawn()` and `MPI_Intercomm_merge()` routines.

The CRAFT library is open source under a BSD license and is available at [12]. Though CRAFT's CR feature is well tested and production proof, its AFT feature (built upon ULFM-MPI) is in its prototype phase and must be used with caution.

8 ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) through the Priority Program 1648 "Software for Exascale Computing" (SPPEXA) under project ESSEX-II (Equipping Sparse Solvers for Exascale) [48]. Special thanks goes to Dr. George Bosilca and Dr. Klaus Iglberger for valuable suggestions and input which helped us overcome design and implementation challenges.

REFERENCES

- [1] M. Snir, R. W. Wisniewski, J. A. Abraham *et al.*, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014.
- [2] J. Dongarra, "Emerging Heterogeneous Technologies for High Performance Computing," <http://www.netlib.org/utk/people/JackDongarra/SLIDES/hcw-0513.pdf>, May 2013.
- [3] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how HPC systems fail," in *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Washington, DC, USA: IEEE Computer Society, June 2013, pp. 1–12.
- [4] J. Hursey, "Coordinated checkpoint/restart process fault tolerance for MPI applications on HPC systems," Ph.D. dissertation, Indiana University, Bloomington, IN, USA, July 2010.
- [5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.
- [6] G. Cao and M. Singhal, "On coordinated checkpointing in distributed systems," *IEEE Transactions on Parallel & Distributed Systems*, vol. 9, pp. 1213–1225, 1998.
- [7] M. Kumar, A. Choudhary, and V. Kumar, "Article: A comparison between different checkpoint schemes with advantages and disadvantages," *IJCA Proceedings on National Seminar on Recent Advances in Wireless Networks and Communications*, vol. NWNC, no. 3, pp. 36–39, April 2014.
- [8] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.
- [9] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "A proposal for User-Level Failure Mitigation in the MPI-3 Standard," http://icl.cs.utk.edu/news_pub/submissions/mpl3ft.pdf, 2012.
- [10] ULFM MPI Standard draft, <http://fault-tolerance.org/wp-content/uploads/2012/10/20170221-ft.pdf>.
- [11] J. Daly, "A model for predicting the optimum checkpoint interval for restart dumps," in *Proceedings of the 2003 International Conference on Computational Science*, ser. ICCS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 3–12.
- [12] F. Shahzad, "Checkpoint/Restart and Automatic Fault Tolerance (CRAFT) library," <https://bitbucket.org/essex/craft>, accessed: 2017-07-27.
- [13] F. Shahzad, M. Wittmann, T. Zeiser, and G. Wellein, "Asynchronous checkpointing by dedicated checkpoint threads," in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 289–290.
- [14] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, Nov. 2010, pp. 1–11.
- [15] W. Bland, K. Raffanetti, and P. Balaji, "Simplifying the Recovery Model of User-level Failure Mitigation," in *Proceedings of the 2014 Workshop on Exascale MPI*, ser. ExaMPI '14. Piscataway, NJ, USA: IEEE Press, Nov. 2014, pp. 20–25.
- [16] S. Pauli, M. Kohler, and P. Arbenz, "A fault tolerant implementation of Multi-Level Monte Carlo methods," in *Parallel computing: Accelerating computational science and engineering (CSE)*, vol. 25. Amsterdam, Netherlands: IOS Press, Sep. 2014, pp. 471–480.
- [17] M. M. Ali, J. Southern, P. Strazdins, and B. Harding, "Application Level Fault Recovery: Using Fault-Tolerant Open MPI in a PDE Solver," in *Parallel Distributed Processing Symposium Workshops (IPDPSW)*, 2014 IEEE International, Phoenix, AZ, USA, May 2014, pp. 1169–1178.
- [18] N. Losada, I. Cores, M. J. Martín, and P. González, "Resilient MPI applications using an application-level checkpointing framework and ULFM," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 100–113, 2017.
- [19] K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using MPI-ULFM," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 51:51–51:56.

- [20] The MPI Fault Tolerance working group, <http://fault-tolerance.org/>.
- [21] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.
- [22] ULFM tutorial and examples at SC'17, <http://fault-tolerance.org/2017/11/11/sc17-tutorial/>.
- [23] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein, "GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems," *International Journal of Parallel Programming*, pp. 1–27, 2016.
- [24] G. G. Sleijpen and H. Van der Vorst, "A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 2, pp. 401–425, 1996.
- [25] M. E. Hochstenbach and Y. Notay, "The Jacobi-Davidson method," *GAMM-Mitteilungen*, vol. 29, no. 2, pp. 368–382, 2006. [Online]. Available: <http://mntek3.ulb.ac.be/pub/docs/reports/pdf/jdgamm.pdf>
- [26] A. Basermann, M. Röhrig-Zöllner, and J. Thies, "The highly scalable iterative solver library phist," in *21th Advanced Supercomputing Environment (ASE) Seminar of the University of Tokyo*, December 2015, funded through DFG Priority Programme 1648 "Software for Exascale Computing", Project ESSEX.
- [27] M. Röhrig-Zöllner, J. Thies, M. Kreutzer, A. Alvermann, A. Pieper, A. Basermann, G. Hager, G. Wellein, and H. Fehske, "Increasing the Performance of the Jacobi-Davidson Method by Blocking," *SIAM Journal on Scientific Computing*, vol. 37, no. 6, pp. C697–C722, Jan. 2015. [Online]. Available: <https://epubs.siam.org/doi/abs/10.1137/140976017>
- [28] Phist library (CRAFT-branch), <https://bitbucket.org/essex/phist/src/craft/>.
- [29] Open MPI bug report, <https://www.open-mpi.org/community/lists/users/2013/10/22832.php>.
- [30] M. Huber, B. Gmeiner, U. Rüde, and B. I. Wohlmuth, "Resilience for massively parallel multigrid solvers," *SIAM J. Scientific Computing*, vol. 38, no. 5, pp. 217–239, 2016.
- [31] Beyond FEAST, <https://bitbucket.org/essex/beast/src/>.
- [32] Fault Tolerance Research at Open Systems Laboratory, <http://osl.iu.edu/research/ft/ompi-cr/>.
- [33] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *The International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.
- [34] "Berkeley Lab Checkpoint/Restart library," <https://ftg.lbl.gov/projects/CheckpointRestart/>.
- [35] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: toward a scalable fault tolerant MPI for volatile nodes," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Nov. 2002, pp. 31:1–31:18.
- [36] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System," University of Wisconsin, Madison, Technical Report CS-TR-1997-1346, Apr. 1997.
- [37] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *Usenix Winter Technical Conference*, January 1995, pp. 213–223.
- [38] R. Fernandes, K. Pingali, and P. Stodghill, "Mobile MPI programs in computational grids," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '06. New York, NY, USA: ACM, 2006, pp. 22–31.
- [39] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," *SIGPLAN Not.*, vol. 39, no. 11, pp. 235–247, Oct. 2004.
- [40] G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo, "CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.
- [41] B. Ramkumar and V. Strumpen, "Portable checkpointing for heterogeneous architectures," in *Symposium on Fault-Tolerant Computing*, 1997, pp. 58–67.
- [42] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2014, pp. 895–906.
- [43] T. N. Ba and R. Arora, "A tool for semi-automatic application-level checkpointing," in *Technical Posters at the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, Aspen, Colorado, Jan. 16–20 2016.
- [44] Open MPI, <http://www.open-mpi.org/>.
- [45] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 7th European PVM/MPI Users' Group Meeting Balatonfüred, Hungary, September 10–13, 2000 Proceedings*. Springer Berlin Heidelberg, 2000, pp. 346–353.
- [46] G. E. Fagg, A. Bukovsky, and J. Dongarra, "HARNESSE and fault tolerant MPI," *Parallel Computing*, vol. 27, no. 11, pp. 1479–1495, 2001.
- [47] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu, "MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware," *Cluster Computing*, vol. 7, no. 4, pp. 303–315, Oct 2004.
- [48] ESSEX project website, <http://blogs.fau.de/essex/>.

Faisal Shahzad is a PhD student in the HPC group at Erlangen Regional Computing Center (RRZE). He received his B.Sc. degree in Mechanical Engineering in 2006 from GIK Institute of Science and Technology, Topi, Pakistan and M.Sc. degree in Computational Engineering in 2011 from University of Erlangen-Nuremberg, Germany. His research area focuses to research various fault tolerance techniques for algorithms in computational science.

Jonas Thies is a scientific employee at the German Aerospace Center, Institute of Simulation and Software Technology, Department of High Performance Computing. He received a PhD in mathematics from the University of Groningen, the Netherlands, in 2010, a Masters Degree in Scientific Computing from KTH Stockholm in 2006 and a Bachelors Degree in Computational Engineering from the University of Erlangen-Nuremberg in 2003. His research interests include sparse matrix computations, HPC software engineering and CFD.

Moritz Kreutzer completed his B.Sc. in Computational Engineering in 2009 and M.Sc. in Computational Engineering in 2011 from the University of Erlangen-Nuremberg, Germany. Currently, he is working as a PhD student in the HPC group at RRZE.

Thomas Zeiser holds a PhD in Computational Fluid Mechanics from the University of Erlangen-Nuremberg. He is now a senior research scientist in the HPC group of RRZE and is among many other things still interested in lattice Boltzmann methods.

Georg Hager holds a PhD in Computational Physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at RRZE. Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His daily work encompasses all aspects of user support in HPC such as lectures, tutorials, training, code parallelization, profiling and optimization, and the assessment of novel computer architectures and tools.

Gerhard Wellein holds a PhD in Solid State Physics from the University of Bayreuth and is a regular Professor at the Department for Computer Science at University of Erlangen. He heads the HPC group at RRZE and has more than 10 years of experience in teaching HPC techniques to students and scientists from Computational Science and Engineering. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.